What is a type?

- a set of values with some common properties and operations on them
  - integers
  - double
  - array of char
  - lists of integers
Types

- **Type annotations:**
  
  - `square :: Int -> Int
    square x = x * x`
    - `square` accepts an `Int` as argument and returns an `Int` as result
    
  - `average :: Double -> Double -> Double
    average x y = (x+y)/2`
    - `average` accepts two `Double` as arguments and returns a `Double` as result
Now to the actual lexer

- **Lexer**
  - Converts a string to a sequence of tokens
  - Strings in Haskell are **lists of characters**
  - How are lists implemented in Haskell?
    - in C, we would use pointers

```c
typedef struct node *link;
struct node {
    int value;
    link next;
};
```
Data Constructors

- **Data constructors** are used to build values of non-basic type

- Lists have **two data constructors**, already predefined in `Prelude` module:
  - `:` right associative infix constructor which takes a data item and a list as argument
  - `[]` the empty list

\[
\begin{align*}
0 &: 1 &: 2 &: [] & \quad & 0 &: (1 &: (2 &: []) ) \\
& & & & & \\
\end{align*}
\]
Programming with Lists

length :: [a] -> Int
length xs = case xs of
    [] -> 0
    (y:ys) -> 1 + length ys

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
Lists

- Since lists are such a central data structure, there is some additional syntactic sugar to make using them more convenient

  - `0:1:2:[]` can be written as `[0, 1, 2]` or any mix of the styles, like `0:[1,2]` `0:1:[2]` (but not `[0,1]:[2]!!`)

  - Strings are lists of characters
  
    - "Hello"
    - `[‘H’,‘e’,‘l’,‘l’,‘o’]`
    - ‘H’:‘e’:‘l’:‘l’:‘o’:[]

```haskell
    type String = [Char]
```

type synonym defined in the Prelude module, type is similar to typedef in C
Lists are everywhere

- Lists are homogeneous - all elements have to have the same type
  - [1, 2, 3] :: [Int]
  - [“hello”, “world”] :: [[Char] or [String]]
  - ['a', 5, 6] type error - Char!
- Many useful higher-order list functions predefined in Prelude (have a look at the module)
  - map :: (a -> b) -> [a] -> [b]
    - map f [x_1, x_2, x_3, x_4] is [f x_1, f x_2, f x_3, f x_4]
  - foldr :: :: (a -> b -> b) -> b -> [a] -> b
    - foldr (+) n [x_1, x_2, x_3, x_4] is x_1+(x_2+(x_3+(x_4+n)))
  - foldl :: (a -> b -> a) -> a -> [b] -> a
    - foldl (+) n [x_1, x_2, x_3, x_4] is (((n+x_1)+x_2)+x_3)+x_4
  - break :: (a -> Bool) -> [a] -> ([a], [a])
    - break (isUpper) "hELlo" is ("h", "ELlo") — isUpper from Data.Char
Back to the lexer

- We now know about the strings, but what about tokens?
- There is no suitable predefined data type
- We need to define our own token type, and data constructors for this type

```haskell
data Token
    = LParen
    | RParen
    | AddOp
    | MulOp
    | SubOp
    | IntLit Int
```

Token is the name we choose for the new type.

LParen, RParen etc are the names we choose for the elements of the new type. They are also called the type constructors of the type Token.
Some other examples

- Days of the week:

```haskell
data Day
  = Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday

-- sample function
isWeekday :: Day -> Bool
isWeekday Saturday = False
isWeekday Sunday   = False
isWeekday day      = True   -- variable ‘day’ matches any day
```
Some other examples

• Shapes:

```haskell
data Shape
    = Square Double           -- square has length as argument
      | Rectangle Double Double -- rectangle has height & width
      | Circle Double            -- circle has radius as argument

-- calculate area of shape
areaOfShape :: Shape -> Double
areaOfShape (Square len) = len * len
areaOfShape (Rectangle height width) = height * width
areaOfShape (Circle radius) = radius * pi * pi -- 'pi' is predefined
```
Type Classes

• What could be the type of the function (==), which compares two data items for equality?

  ‣ (==) :: a -> a -> Bool  no, that’s too general!

  ‣ (==) :: Eq a => a -> a -> Bool

  ‣ if a is a member of type class Eq, then (==) can compare two values of this type for equality

  ‣ when we define a new data type, we can include it into the class using deriving

```haskell
data Token
  = LParen
  | RParen
  | AddOp
  | MulOp
  | SubOp
  | IntLit Int
deriving (Eq, Show)
```
List processing

- List processing is easy

- Let’s write a program that adds all the elements of a list
  - type class of numeric types: `Num`
  - to traverse list recursively, we can use pattern matching:

```haskell
addUp :: Num a => [a] -> a
addUp [] = ....
addUp (x : xs) = .....```

Syntactic Peculiarities

• Case matters:
  ‣ variable and function names must start with lowercase
  ‣ data constructor, type constructor, and class names must start with uppercase

• Indentation matters:

```plaintext
average x y = xy / 2
where
  xy = x + y  ✓ ok
```

```plaintext
average x y = xy / 2
where
  xy = x + y  ✗ syntax error
```

• The language allows the use of braces and semicolons instead, but that’s not commonly used
GHCi

• running the compiler for every small program or change is tedious
• GHCi is an interactive Haskell interpreter environment
• can mix compiled and interpreted code
• additional functionality
  ‣ infer and print types
  ‣ print information about identifiers
  ‣ integrated, source level debugger
Types in Haskell

• Basic types:

★ Int Integer Float Double Char ....

• Composite types:

★ Tuples: (Int, Float) (Char, Int, Double)

★ Lists: [Int] [[Float]] [[(Char, Float)]]

• New name for existing types

★ a standard definition from the Prelude:

    type String = [Char]
Types in Haskell

• Enumeration types:

★ simple form of algebraic data type:

```haskell
data Bool = True | False
  deriving (Show, Read, Eq, Ord)
```

• Algebraic types:

★ like enumeration types, but data constructors can have arguments:

```haskell
data Shape = Circle Float -- radius
            | Rectangle Float Float -- length, width
  deriving (Show, Read, Eq, Ord)
```
Types in Haskell

• Recursive data types

  ➤ in C:

```c
typedef struct listNode * link;

struct listNode {
    int item;
    link next;
};
```

```c
typedef struct treeNode * link;

struct treeNode {
    int item;
    link leftTree, rightTree;
};
```

➤ in Haskell

```haskell
data IntList = Cons Int IntList |
               EmptyList
```

```haskell
data IntTree = Node Int IntTree IntTree |
               EmptyTree
```

★ how do we create a list, tree in a C, Haskell program?
Haskell

• Most widely used implementation
  ‣ Glasgow Haskell Compiler (GHC)
  ‣ Open source (BSD3)
  ‣ available for most popular OSes (GNU/Linux, MacOS, Windows, Solaris, etc)
  ‣ is both a highly optimising compiler as well as an interactive interpreter
  ‣ implements many extensions to Haskell ‘98
  ‣ comes with a extensive set of libraries (with many more at http://hackage.haskell.org)
  ‣ support for multicore parallelism
Infix operators

• Usually, a function application has the form
  - `functionName arg1 arg2 ....`

• but some are defined as infix operators, for example
  - `arg1 + arg2`

• We can use infix operators in prefix notation if we put them in parenthesis
  - `(+) arg1 arg2`

• and prefix functions as infix by using single quotes
  - `7 `div` 3`
  - `div 7 3`

• you can also define your own infix operators, with precedence and associativity
Partial function application and Currying

• Why the strange notation for functions with multiple arguments? e.g.,

  ‣ (+) :: Num a => a -> a -> a and not, as we might expect
  ‣ (+) :: Num a => (a, a) -> a

• The function type constructor -> is right associative, so

  ‣ a -> a -> a is actually the same as
  ‣ a -> (a -> a)

• that is, (+) is can been seen as a function, which accepts one numeric value as argument, and returns a new function!

• it is perfectly fine to write apply (+) to only one argument (partial application)

• any function of type a -> b -> c can be mapped to an equivalent function of type (a, b) -> c and vice versa
Higher-order functions

• **Traversal patterns:**
  - incList: increment all elements of a list
  - this traversal pattern is very common - can we generalise it?

• **Can we generalise addUp?**
  - many functions share the same traversal pattern
  - let’s extract this pattern

• **Higher-order functions** are functions which either take another function as an argument, or return a function as result
More about Type Classes

• A **type class** in Haskell is a set of types which share a number of operations.

• Not to be confused with classes in OO languages (more like Java interface).

• Examples:

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

class (Eq a) => Ord a where
    (<)  :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (> ) :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    max  :: a -> a -> a
    min  :: a -> a -> a

class Show a where
    show :: a -> String
    ...
```
More about Type Classes

• Two ways to include a user defined type in a type class

  ‣ Method (1): use **deriving**; only works for some predefined, frequently used type classes like Eq, Ord, Show, Read

    ```haskell
    data MyShinyNewType
    = This Int
    | That String
    deriving (Eq, Ord, Show)
    ```

  ‣ Method (2): the programmer explicitly provides the definition for the member functions of the class

    ```haskell
    data MyShinyNewType
    = This Int
    | That String

    instance Eq MyShinyNewType where
    (==) (This n) (This m) = True
    (==) _ _ _ _ _ _ _ _ _ _ _ _ = False
    (/=) t1 t2 = not (t1 == t2)
    ```
Monad and contextual information

• The behaviour of functions in a pure functional language does not depend on a global state

• However, we all know from experience that sometimes we have a complex state which we don’t want to pass explicitly to all the functions that depend on it

• For I/O functions, this wouldn’t even be possible, since the state their behaviour depends on includes not only the state of hardware, but also the user

• Monads encapsulate the state and “hide” it from the user
Monad

• I/O in C:

```c
int compare_chars () {
    int a, b;

    a = getchar();
    b = getchar();
    return (a < b);
}
```

• I/O in Haskell:

```haskell
compare_chars :: IO Bool
compare_chars = do {
    a <- getChar();
    b <- getChar();
    return (a < b); -- return :: a -> IO a
}
```

• A function of type IO a

  ‣ performs an operation depending and/or altering the state of the world

  ‣ when done, returns a value of type a
Monad

So, if programming in the IO Monad is like programming in C, why bother?

★ **Advantage:** control side effects

- Different signatures, different properties:
  
  ```haskell
  noSideEffect :: Int -> Int  
  maybeSideEffect :: Int -> IO Int
  ```

- checked by the compiler, simplifies debugging

- required for concurrency:

  ```c
  int compare_chars () {
      return (getchar () < getchar ());
  }
  ```